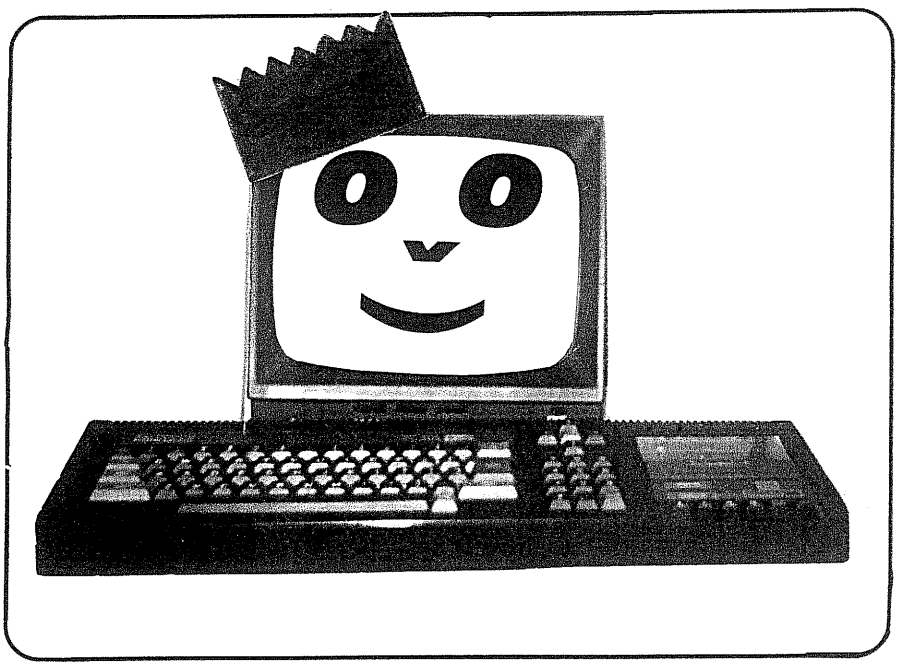
ISSUE 7

PRINT—OUT

Price 70p

BIRTHDAY NUMBER



Written by Thomas Defoe and Mark Gearing
Contributors are Bob Taylor and Alan Scully

INCLUDING:

HOMEBREW
BASIC EXPLORED
RSXS - PART 3
DISC NAMER
MACHINE CODE

INDEX—No.7

Miscellaneous

Features

Reviews

Programming

We would like to express our thanks to Mr. Gearing and Black Horse Agencies Januarys for the continued use of their photocopier in producing Print-Out.

Editorial

WELCOME TO ISSUE SEVEN OF PRINT-OUT !!!

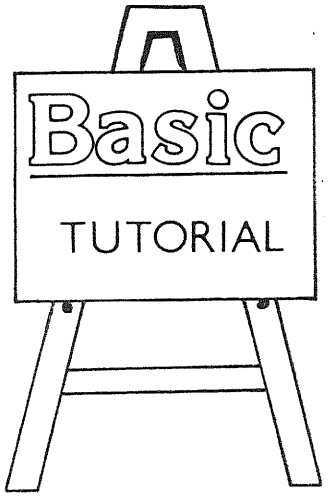
As you may have noticed, there has been a major change to Print-Out as now, after six issues, a regular name no longer features on the front of the magazine. Unfortunately, we have had to say goodbye to Jonathan Haddock, who has decided to leave Print-Out. None of the authors work full-time on the magazine but rather do it when they have the time. And as Jon has been finding that other occupations require more and more time in recent months, he has decided that he is unable to continue writing for Print-Out. Although, at present, we have no actual replacement for him, we hope that the magazine will run as smoothly! We all wish Jon the best of luck in the future.

We hope that you enjoy this issue of Print-Out, and remember that you can order a copy of Issue Eight of Print-Out in advance - see page 38 for more details on this & other offers.Please take this opportunity to fill in the prize questionnaire on page 42 as your opinions and ideas are very highly valued.

If you've any queries or problems with the CPC, please write to us at the address below & we'll do our best to solve your problem. We guarantee that all letters will be answered personally by one of the writers of Print-Out. The address is the same for all orders, and is printed below:

PRINT-OUT, 8 Maze Green Road,
Bishop's Stortford, Herts CM23 2PJ.

COMPETITION→p.41



BEGINNERS BASIC

SOUND

As was mentioned last issue, there have been some major changes to the way in which the BASIC programming sections are now arranged and there are a couple of further alterations to come. Unfortunately not all of the BASIC articles were ready in time for this issue & so we've had to include rather more Machine Code items than we would normally. Well, next issue will see the start of several new features on BASIC. In the meantime, Beginner's BASIC will carry on investigating various commands and 'Advanced BASIC' will continue its examination of the BASIC Operating System and Parser.This month, we're going to look at the SOUND command in its simplest form and also what it does.

At first glance, the SOUND command may seem a complicated beast and a very daunting prospect. The manual, in its usual helpful way, does nothing to dispel this belief and the description of the SOUND keyword as

```
SOUND <channel status>,<tone period>[,<duration>[,<volume>[,<volume envelope>
                                    [,<tone envelope>[,<noise period>]]]]]
```

does not really help matters. All of the things in square brackets, however, are optional and can be omitted. Hence the SOUND command becomes far more manageable when it is in its simplest form of SOUND <channel status>,<tone period>.

CHANNELS

Before going any further, I have to confess that I am no musician and know virtually nothing about composing, melodies, etc!! However, I do know how to get the CPC to play something fairly close to music. First, let me explain what the 'channel status' is and why we have it.

When you have a chord, it involves playing more than one note at a time and the CPC uses its channel facility to allow it to do this. On the computer you've three channels, labelled A, B and C for reference, and each of these can be told to play a different note from each other - they can of course also play the same note. It might help to think of the channels as being three separate instruments although this is not strictly true (as they are all produced by one sound chip). These channels can be linked or separated in different ways but for now, we will just consider their straight forward use. Each channel has a number that is associated with it, & this number tells the CPC which channel to play the note on.

This number is the channel status and the values we are looking at are:

1 Channel A only 2 Channel B only 4 Channel C only

Thus, to play a note on channel A only, the channel status would be 1. The next thing to look at is the 'tone period'.

The Tone Period is basically a fancy name for the pitch of the note or, in other words, which note it is that we wish to play. The various values, and the notes they refer to, are all given in the back of the manual. The reasons as to how these particular values are derived are also given but I wouldn't worry too much as all of the information needed is printed in one of the appendices. Just bear in mind that the higher the number is, the lower the note produced.
If we look up the data for 'Middle C' we find the following information:

NOTE	FREQUENCY	PERIOD	RELATIVE ERROR	
C	261.626	478	+0.046%	Middle C

You can ignore everything except the name of the note (C) and the period (478). It's the period number which we have to put in the sound command to produce the required note. So, to play Middle C on channel A, type: SOUND 1,478
The channel number for A is 1, and the period value for Middle C is 478. To get International A played, simply change the period value: SOUND 1,284

That is all there is to the SOUND command in its simplest form. At present, it is not very versatile & we would have a hard time trying to produce anything vaguely musical. If you listen to any note you will realise that it has several characteristics. One of these is its pitch (which we have already covered), the others are its loudness (volume) and its length (duration) and these are two of the optional parameters of the SOUND command.

DURATION

The first optional item that we come to, is the 'duration' ie the length of the note. When this is added, the definition of the SOUND command becomes:

SOUND <channel status>,<tone period>[,<duration>]

The duration of a note is measured in units which are 1/100th of a second long. This is all there is to the duration part of the command, and so to play Middle C for one second on channel A, we would use: SOUND 1,478,100

If we do not specify a 'duration', the length of the note is automatically assumed to be 20/100ths of a second (ie 0.2 seconds). However, when we are using a volume envelope as well, there are complications. But as we haven't dealt with this yet, we'll just forget about it for the time being.

There is just one other thing to notice concerning the duration command. In real music, you often see two of the same notes following each other. The musician would play these notes with a short gap in between, in order to distinguish

them from one another. Unfortunately, the computer does not. If you type in the following line SOUND 1,478,100:SOUND 1,478,200 you might expect there to be two notes of Middle C played one after another (the first lasting 1 second and the second lasting two seconds). Instead you will get one note lasting three seconds!!!. To make the distinction between the notes, you need to play a silent note (ie a rest) and this leads us onto the next section, volume.

The next stage is the note's volume, and this is where things can get a bit tricky - due mainly to a difference between the 464 and the 6128. On the 464, we can have volume in the range of 0 to 7 (with 7 being the loudest and 0, silent). However, with the 6128 we get the same range of volume but with different values - now 0 is silent and 15 is the loudest. Thus volume 15 on the 6128, corresponds to volume 7 on the 464.

Therefore to play middle C on channel A for 1 second at maximum volume, we would use (for the 464): SOUND 1,478,100,7

(and for the 6128): SOUND 1,478,100,15

This is further confused when we look at volume envelopes but don't worry about them for the moment.

Going back to the problem we had with two notes running into each other, we now have the solution. To play the two notes of Middle C with a break in between we would use: SOUND 1,478,100,7:SOUND 1,0,1,0:SOUND 1,478,200,7

The pitch of the middle note does not matter as it is being played at volume 0, and will never be heard. All it does is delay the second note by a short period of time (in this case 1/100th of a second) and thus giving the effect of playing two separate notes.

That's it for this issue, but next time I'll be looking at the various tone and volume envelopes in a special section which is dedicated to BASIC sound programming and Beginner's BASIC will also be looking at some more common keywords.

-The Firmware

VITAL READING ON M/CODE

It has been quite a while since Amstrad decided to discontinue the Firmware Manual for the CPC and few books have been produced to take its place. For those of you who have never heard of 'The Firmware Manual', here is a brief summary:

Quite simply, the Firmware Manual is *the* book about how and why the Amstrad works; it's invaluable to any Machine Code programmer or CPC enthusiast. However the manual is definitely not for the faint-hearted as it is written in technical jargon. Still, the amount of information that it contains is incredible. In its many pages it covers topics ranging from ROM expansion to driving the sound chip. One of the most important sections is its description of the Firmware Jumpblock, which tells you how to correctly use the Machine Code calls.

Unless you intend to access the lower ROM directly (not a good idea!), any program which you write in Machine Code will use these firmware calls and their entry and exit conditions are essential. As there are over 200 of these 'calls', it is unlikely that you will be able to remember all of the things that need to be done before a certain routine is used; and this is where the Firmware Manual comes in very useful.

Now that it has been discontinued, the best book available is the 'Amstrad Advanced Users Guide' by Daniel Martin (publisher: Glentop ISBN 1-85181-122-2). However Print-Out intends to provide another option. Over the next half a dozen issues, we hope to be able to print all of the firmware calls together with the entry and exit conditions and a brief description of what it does.

First, here is a summary of what information each entry includes. They all appear in the following form:-

NUMBER ADDRESS to call NAME of the routine
BRIEF DESCRIPTION - what the routine will do when it is called
ENTRY CONDITIONS - what has to be done before it can be called
EXIT CONDITIONS - what will have happened to the registers when the routine has finished being executed
SPECIAL NOTES - this part is not always present

The exact number of issues that this will take up has not yet been decided, but it should give anybody who wishes to progress further with Machine Code enough information to do so. So here goes with this mammoth task.

000 &BB00 KM INITIALISE
ACTION: This routine will initialise the Key Manager and everything will be set up as it is when the computer is first swtiched on.
ENTRY: No entry conditions
EXIT: AF,BC,DE,HL will be corrupted; all others preserved

001 &BB03 KM RESET
ACTION: This resets the Key Manager (especially indirections and buffers)
ENTRY: No entry conditions
EXIT: AF,BC,DE,HL will be corrupted; all others preserved

002 &BB06 KM WAIT CHAR
ACTION: Waits for the next character from the keyboard
ENTRY: No entry conditions
EXIT: Carry flag is set to true; A holds the character value; flags are corrupt; all others preserved

003 &BB09 KM READ CHAR
ACTION: Tests to see if a character is available from the keyboard
ENTRY: No entry conditions
EXIT: If character was available - carry true; A contains character
OTHERWISE - carry false; A corrupt; ALWAYS - others preserved

004 &BB0C KM CHAR RETURN
ACTION: Save a character for the next use of KM WAIT CHAR or KM READ CHAR
ENTRY: A contains the ASCII code of the character to be put back
EXIT: All registers preserved

005 &BB0F KM SET EXPAND
ACTION: Assigns a string to a key-code
ENTRY: B holds the key-code; C holds the length of the string; HL contains the address of the string
EXIT: IF OK - carry true; ELSE - carry false; ALWAYS A,BC,DE,HL corrupt

006 &BB12 KM GET EXPAND
ACTION: Reads a character from an expanded string of characters
ENTRY: A holds an expansion token (a key-code); L holds a character number
EXIT: IF OK - carry true; A holds character; ELSE - carry false; A corrupt
ALWAYS - DE corrupt; all others preserved

007 &BB15 KM EXP BUFFER
ACTION: Set aside a buffer area for character expansion strings
ENTRY: DE holds the address of buffer; HL holds the length of the buffer
EXIT: IF OK - Carry true; ELSE - carry false; ALWAYS A,BC,DE,HL corrupt

008 &BB18 KM WAIT KEY
ACTION: Waits for a key to be pressed
ENTRY: No entry conditions
EXIT: Carry true; A holds character; all other registers preserved

009 &BB1B KM READ KEY
ACTION: Test whether a character is available from keyboard; does not wait
ENTRY: No entry conditions
EXIT: IF available - Carry true, A contains character; ELSE Carry false,
A corrupt; ALWAYS - other registers preserved

010 &BB1E KM TEST KEY
ACTION: Test if a particular key (or joystick) is pressed
ENTRY: A contains the key/joystick number
EXIT: IF pressed - Zero false; ELSE - Zero true; ALWAYS - Carry false; A and HL corrupt; C holds SHIFT/CTRL status; all others preserved

011 &BB21 KM GET STATE
ACTION: See state of SHIFT LOCK and CAPS LOCK
ENTRY: No entry conditions
EXIT: If L holds &FF then SHIFT LOCK is on; if off, L holds &00
If H holds &FF then CAPS LOCK is on; if off, H holds &00

MAC II ~ by Alan Scully

Mac II was a very professional effort and one that I am sure is better than some of the budget software which you can buy from the shelves. In fact the only noticeable omissions were the lack of a title screen and music. These apart, the game was virtually faultless.

The aim of the game is to travel through each level collecting treasure and avoiding the ghosts. In every maze, there are five treasures to collect but they are hidden behind locked doors and so it's necessary to get the keys first. Each maze is quite large and with ten of them on the disc, plus the ability to design more of your own, there is plenty to keep you entertained.

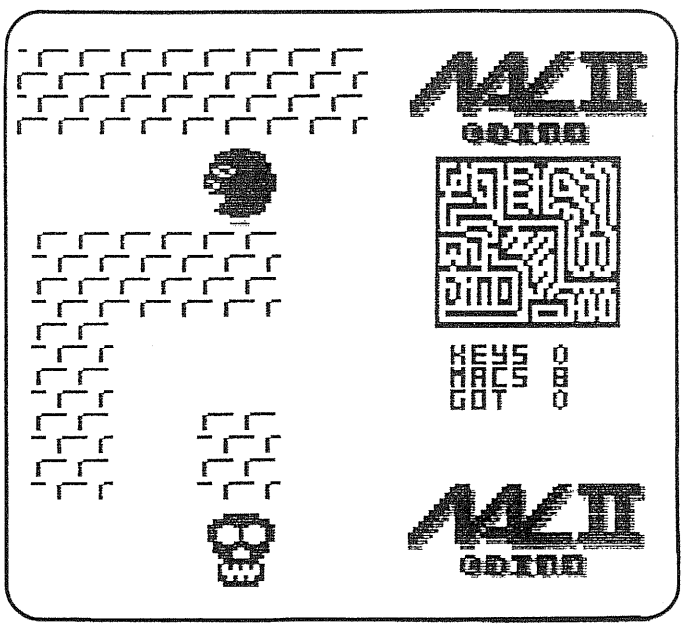
Although there is no background music, the multi-colour graphics certainly make up for it. These are truly excellent and are the best I have ever seen in a homebrew game. They're crisp and clear and make the game seem full of quality.

The game can be played with keys or joystick, & it presents the player with quite a challenge but I found that it was not so difficult that I got frustrated immediately. The mazes contain thirty screens, and each level is harder than the ones before.

The actual layout was well thought out and this gave an uncluttered display. The only problem with the actual game was that only a small part of the maze was shown at a time and you couldn't see where the exits onto the next screen were - this made the map vital. Even though the scrolling was very good, the map took a while to draw at the beginning of each maze.Despite the lack of any music, there were some sound effects which added to the atmosphere.

There were some 'extras' which are seldom found in full-price software.One of these was a Level Designer: it ought to have added to the game's appeal but, while it was a very good idea it wasn't terribly well implemented although with more thought it could have been so much better.The instructions are supplied on the disc & could be printed out if required.You could also switch the map off, as well as altering the number of lives to suit your ability.

The game was good fun and I'm sure that many people will enjoy playing it. This program is available from Scull PD (see the article on page 18 for prices/ details) so it shouldn't break the bank. The address is 119 Laurel Drive, Greenhills, East Kilbride, Glasgow G75 9JG.



Avoiding the Ghost

MAC III ~ by Alan Scully

Mac III, the second game from Alan for the moment, was again good although not nearly as professional as Mac II. Having said this, the game had many points of merit.

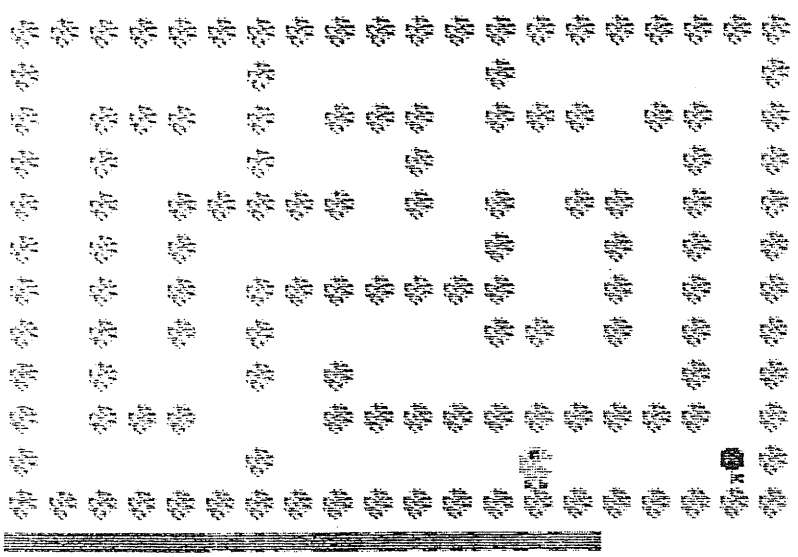
It is a bit like 'Pac-Man' in style and layout but this time you control a small object, called Mac, who has to collect several items from the screen. What makes it difficult is that the screens have to be completed in a certain time.

The game sounds simple in theory but in practice it is really quite tricky. Mac has an unusual tendency to bounce off the wall and fly back in the direction where he came from. While this made the game more interesting, the response was not particularly good - thus making timing difficult.

Despite this, the actual screens are clear, simple and the graphics, whilst not being particularly detailed, were varied and were good to look at. There was a total lack of music and the sound effects were only rudimentary.

The game could have been considerably improved if there were more things to pick up and if the screens were different. It would also have been more exciting and unpredictable if there had been a longer time available, and if a few ghosts had been included for you to avoid.

However, there was enough in the game to keep you busy for a while, and was fairly addictive. After a time, the screens became a bit repetitive and the game tended to get tedious. The plot, whilst not specially original, has some unusual twists which make for a more interesting & enjoyable game than your average Pac-Man clone.



Getting the Key

In conclusion then, Mac 3 is a very good game which could have been much better by the addition of one or two frills. It is still a most enjoyable game for several hours relaxation & when you consider that you can buy both Mac II and Mac 3 (plus a few more games) for just £1, they represent truly excellent value-for-money and are a wonderful introduction to Scull Public Domain.For further details of this & other bargains from his PD library turn to the article on page 18 for more information) and his address is;

119 Laurel Drive, Greenhills,
East Kilbride, Glasgow G75 9JG.

REVENGE OF CHAOS

by Tony Kingsmill

Revenge of Chaos is the follow up to the adventure game 'Island of Chaos', which I reviewed favourably in Issue Four of Print-Out. The game keeps the same basic format as its predecessor - all that changes is the story line.

In 'Island of Chaos' you had to bring about the death of the evil warlord, Baktron, but now he's been reincarnated by powerful magic and he wants revenge. As part of his plan, he destroyed the city of Brael Ti, killing its inhabitants instantly. This was just the beginning as his next step was to be much worse...

In the adventure, you play the part of the leader of a group of paladins & must try to step Baktron from taking over the world. Unfortunately, as luck has it, the other paladins have a 'pressing engagement' elsewhere & leave you alone on the island.

The game itself is written using the Quill and the graphics were designed with the Illustrator. I felt that, although it is the sequel to Island of Chaos, it was just too similar and so there is not much difference in standard between the two programs. The adventure had a good loading screen and fairly descriptive locations.

As before, the graphics weren't particularly brilliant but there was a noticeable improvement from last time. The usual 'save' and 'load' functions, which are vital with a game like this (it boasts over 76 locations), were incorporated into the program, along with a 'help' option. Tony also offers a free hint sheet for Revenge of Chaos if you get really stuck, and this is also the case with his other games.

The presentation of the game was again good and this gave it a professional appearance. As for addictiveness, it had some appeal but I felt that it was just to similar to other games from the same stable - this should be a plus point for Tony's previous customers! As the text was the only thing that had changed, this might mean that some people would find the game a bit boring, but, if you are an ardent adventure enthusiast, this should be just the thing (as you're guaranteed hours of puzzlement at an excellent price.

The game costs £3.95 on disc only and comes with a free game, Alien Planet. This is in the same mould to Revenge of Chaos, although smaller, and so provides more entertainment and challenges for the adventure player.

If you don't own a disc drive and are sorry to be missing out on the games, Tony plans to produce an 'Adventure Compilation' later in the year which will be available on tape and disc at 'a very reasonable price'. This will include some of Tony's previous releases, as well as the possibility of a new adventure game or two. Of course, we'll keep you informed of any further developments.

In the meantime, here is Tony's address;

202 Park Street Lane, Park Street,
St. Albans, Hertfordshire AL2 2AQ.

ADVANCED M/CODE

An introduction to **RSXs** *(part 3)*

RE-LOCATING YOUR CODE

by Bob Taylor

Often there is a need to be able to load RSXs into any reasonable area of RAM and not at one predetermined location (the area usually used is just above HIMEM which will itself have been moved to make room for the RSX). In order to do this some form of re-location routine will be needed to adjust any absolute addresses which occur in the routines, and move with them. We will not need to alter any address not within the code that's being moved. (Relocation could be achieved from BASIC but, since this is an article about Machine Code programming, we will concentrate on self-relocation using this medium) In an RSX initialisation routine two such address alterations are usually required: those of the Command Table & of the Command Name Table. Two further locations will need to be ascertained although we can eliminate the requirement for obtaining their addresses.

We are helped here by an undocumented facet of the Operating System which results in the DE register holding the address of entry to a CALLed routine on entry to the routine. (This only holds true if the CALL, from BASIC of course, isn't accompanied by any parameters.) Any absolute addresses that are required will have an offset to this entry point, and so if we add this offset for each occurrence to the base address, we can obtain the new address needed.There are two main ways of doing this; which one we choose depends on how many addresses we have to alter.

---

1. If there are few alterations to make we can actually load the BC or DE registers with each offset and add these to the entry address held in HL. The next routine given gives one method of achieving this:

```
.intrsx PUSH DE          ;-> (1) Entry Point   (I use these numbers alongside
                         ; all PUSH and POP instructions to show the depth of
                         ; entries on the Machine Stack.)
.datblk LD   H,D         ;the 3 instructions starting here will not be needed
                         ; again so the four bytes involved can be reassigned
                         ; for use as the chaining block.
        LD   L,E         ;HL now has the entry address also.
        LD   (HL),&C9    ;load the entry point with a RET instruction so that
                         ; the routine cannot be CALLed a second time.
        LD   BC,namtbl-intrsx
        ADD  HL,BC       ;HL-> .namtbl.
        EX   DE,HL       ;DE-> .namtbl; HL-> point of entry.
```

(cont.)

(cont.)

```
        LD    BC,comtbl+1-intrsx
        ADD   HL,BC       ;HL-> byte after .comtbl.
        LD    (HL),D      ;
        DEC   HL          ;
        LD    (HL),E      ;load .comtbl DEFW with .namtbl address. HL-> .comtbl.
        LD    B,H
        LD    C,L         ;BC-> .comtbl for KL LOG EXT Firmware routine.
        POP   HL          ;Entry Point (0) (now no entries left on the stack).
        INC   HL          ;HL-> .datblk for KL LOG EXT. We're using a redundant
                          ; part of this initialisation routine for .datblk.
        JP    &BCD1       ;to KL LOG EXT Firmware routine.

.comtbl DEFW namtbl       ;address used by Firmware RSX handling routine.
.rsx                      ;assuming one RSX only, it can start here.
        etc
```

NOTE: We arranged things so that we didn't have to calculate the address of the Data Block, nor of the entry point for storing the RET byte. Of the two addresses we did calculate, one was the address of where to store the other.

---

2. This last feature is also common to the second method of re-location and it uses a table of the offsets instead of loading registers directly with them. Sometimes the table contains both sets of address offsets; ie where to store and what to store. This is however unnecessary and only the 'where to store' offsets need be there, the 'what to store' ones being at their destinations already, waiting to be picked up, added to, and then re-stored again & thus saving two bytes per re-locatable address.

The RSX routine later in this article has this method of re-location incorporated into its initialisation routine & shows the 'where to store' offsets being picked up from a table to create addresses from which existing offsets are then picked up, converted and re-inserted.

NOTE: By making '.comtbl' follow on from the last entry in the offset table, we do not have to calculate its address to load into BC for KL LOG EXT - instead we just pick up the value of HL which now points to .comtbl.

NOTE: The second offset table entry has a further offset added: the +1 makes the two address bytes the object of the calculations and not the LD HL instruction itself. This will apply to most addresses which need altering; however, with 4 byte instructions, like some for IX and IY & also some for BC, DE and SP, this extra offset will need to be +2.Only when an address is present as a DEFW will no secondary offset be needed.

The entries in the offset table can be in any order. Usually there will be more entries required than are shown in the example which has been pared for maximum efficiency. The addresses to which these entries point have got to be stored as

'address-intrsx' in each case (eg as at .add1). It helps to have as few address changes as possible, leading to a smaller offset table. We've seen in previous articles that by using a JR instead of a JP in the Command Table we can reduce the number requiring adjustment and we have already done so here.This can also be applied to JPs within the RSX routine itself (although ours is too short to have any) by replacing with two or more JRs conveniently placed throughout the code to enable the Program Counter to skip from one to another until the wanted destination is met. Since this increases the length of the code and also slows it down slightly (a JR takes two 'T' states longer than a JP, and we are using at least two JRs instead of one JP), it should only be used where time constraints allow and where a slight increase in room is tolerable.On the other hand it may be useful if the need for the extra bytes used by the longer re-location routine can be eliminated by having no extra addresses to adjust.

In the case of sub-routines whose CALL (from M/C) addresses need re-location, it is not possible to use JRs of course.

Sometimes it's possible to write the code in a different way & eliminating the CALLs.If the subroutine is used only once, it should be possible to incorporate it into the main routine at the point at which it was CALLed and so eliminate the address that way.This has been done with the message printing routine below which has then been fine tuned for optimum efficiency. Similarly, if it's called only a few times and it is very short, it might be worth while inserting copies of it where it was CALLed and so again get rid of the routine altogether. All of these things have to be weighed up at the time of writing the code and a decision made then as to the best approach.

READ AND WRITE RSX

The following routine provides two new RSXs comprising a WRITE for placing strings in memory with a corresponding READ for retrieving them. The initialisation routine with which it starts incorporates the second method of relocation as mentioned above:

```
.intrsx LD   HL,addtbl-intrsx ;note that .datblk is located at the second byte
                              ; of this instruction.
        ADD  HL,DE      ;HL-> addtbl. (DE holds the entry address to .intrsx)
        LD   B,&02      ;2 offsets of addresses -> addresses to relocate.
.nxtadr LD   A,(HL)     ;get LB of first/next offset of address.
        INC  HL
        PUSH HL         ;-> (1) TBLPOS (Address Table position reached).
        LD   H,(HL)     ;get HB of offset.
        LD   L,A        ;HL= offset.
        ADD  HL,DE      ;HL= address of address; ie HL->address to relocate.
        LD   A,(HL)     ;get LB of this address.
        ADD  A,E        ;add in LB of routine entry point.
```

```
        LD   (HL),A     ;re-insert corrected LB of address.
        INC  HL
        LD   A,(HL)     ;get HB of address.
        ADC  A,D        ;add in HB of entry point + any carry from LB addition.
        LD   (HL),A     ;re-insert corrected HB.
        POP  HL         :TBLPOS (0)
        INC  HL         ;step on to next offset.
        DJNZ nxtadr     ;if more offsets - otherwise HL-> .comtbl
        LD   B,H
        LD   C,L        ;BC-> .comtbl for KL LOG EXT.
        EX   DE,HL      ;HL-> entry point.
        LD   (HL),&C9   ;store RET at entry point to stop re-running .intrsx.
        INC  HL         ;HL-> byte after entry point to be used as Data Block
                        ;for KL LOG EXT.
        JP   &BCD1      ;to log these RSXs on.
.namtbl DEFB "WRIT","E"+&80
        DEFB "REA","D"+&80
        DEFB &00
.addtbl DEFW comtbl-intrsx
        DEFW add1+1-intrsx
.comtbl DEFW namtbl-intrsx ;will become address of Name Table after re-location
.write  SCF             ;signal 'WRITE'.
        DEFW &0100      ;dummy instruction: absorbs the OR A byte following so
                        ; eliminating clearing the Carry flag, being treated as
                        ; LD BC,&B700; next instruction is first DEC A. 2 'T'
                        ; states quicker than using JR 'first_dec_a'.
.read   OR   A          ;clear Carry: signal 'READ'.
        DEC  A          ;on entry, A held the number of parameters
        DEC  A          ;DEC A twice does the same as CP &02 but maintains the
                        ; Carry Flag (with a penalty of three extra 'T' states).
        JR   Z,getstr   ;Zero set if two parameters (READ or WRITE).
        JR   C,parerr   ;if WRITE and not 2 parameters.
        DEC  A
        JR   NZ,parerr  ;if READ and > 3 (or <2) parameters).
        INC  A          ;clear Zero Flag to signal Length parameter.
        INC  IX
        INC  IX         ;step on to string parameter. IX displacements are now
                        ; identical whether 2 or 3 parameters were entered. No
                        ; further alterations to the IX base address are needed.
.getstr LD   H,(IX+1)   ;get 2nd parameter -
        LD   L,(IX+0)   ; HL=addr of string descriptor.
        LD   A,E        ;on entry to a routine or RSX with parameters, DE will
                        ; hold the last parameter. Assume READ with Length
                        ; parameter so save Length in A; it is not necessary to
                        ; use high byte (which would be 0 anyway). If not Length
                        ; then A's contents will be unused.
```

```
        LD   C,(HL)     ;get string length from descriptor.
        INC  HL
        LD   E,(HL)
        INC  HL
        LD   D,(HL)     ;DE=addr of start of string (from descriptor).
        PUSH AF         ;-> (1) Carry/Zero Flags and possible Length.
        DEC  DE
        DEC  DE         ;step back to 'length' byte before string proper.
        LD   A,(DE)     ;get this 'length'.
        CP   C          ;Zero if 'length' byte and descriptor length match.
        JR   NZ,strerr  ;if no match then definitely not a string.
        INC  DE
        INC  DE         ;step on again to start of string.
        LD   H,(IX+3)   ;get 1st parameter -
        LD   L,(IX+2)   ; HL=addr to READ from, or WRITE to.
        EX   DE,HL      ;assume WRITE so DE->addr to WRITE to; HL->string start.
        POP  AF         ;Carry/Zero Flags and possible Length parameter (0).
        JR   C,move     ;if WRITE then DE, HL and string length set correctly -
        EX   DE,HL      ;otherwise DE->string for result; HL->addr to READ from.
        JR   Z,move     ;if READ and no Length parameter.
        CP   C          ;compare Length parameter with length of string - use
                        ; whichever is shortest.
        JR   NC,move    ;if Length is greater then use string length (in C) -
        LD   C,A        ;otherwise move Length into C.
.move   LD   A,C
        OR   A
        RET  Z          ;if Length/string length is 0 then do nothing.
        LD   B,&00      ;BC=length to transfer (for LDIR).
        LDIR            ;transfer to string if READ, or from string if WRITE.
        RET             ;to BASIC.
.parerr LD   B,1        ;signal 1st error message.
        DEFB &11        ;dummy line: combines with the next two bytes to appear
                        ; as LD DE,&0206 if entering at .parerr so avoiding
                        ; re-loading B with 2. Next instruction is LD C,B.
.strerr LD   B,2        ;signal 2nd error message.
        LD   C,B        ;save the message number.
.add1   LD   HL,errmes-intrsx ;after re-location, HL will -> error messages
                              ; starting with 'Check Parameters'.
        JR   prtmes     ;into the message printing routine.
.nxtchr LD   A,(HL)     ;get first/next character.
        INC  HL         ;step on to following char.
        OR   A          ;a byte of &00 is used to end a message string.
        JR   NZ,nxtchr  ;if not the end of the string.
.prtmes DJNZ nxtchr     ;enter routine here; 1st error message goes straight
                        ; into .prtstr, 2nd goes via .nxtchr to find end of 1st
                        ;message.
```

```
.prtstr A,(HL)          ;get first/next char. of string to print.
        OR   A
        CALL NZ,&BB5A   ;print char if not the end marker; all registers and
                        ; flags preserved.
        INC  HL         ;step on to next char.
        JR   NZ,prtstr  ;for next character if not end of message marker.
        LD   A,32
        ADD  A,C        :ERR number; gives 33 for parerr & 34 for strerr.
        LD   C,0        ;for ROM 0 (BASIC).
        LD   HL,&CB55   ;(&CA93 for 464); BASIC's Error Handling routine in ROM.
        JP   &001B      ;to FAR CALL to ROM; the Error Handling routine alters
                        ; the stack so doesn't REturn.

.errmes DEFB "Check Parameters",&00
        DEFB "Check String",&00
```

Although not obvious initially, the routines for WRITE and READ have much in common and so have been amalgamated into one, using the Carry Flag to indicate where the two routine paths differ from each other.

HOW TO USE THE RSX

The full syntax of each RSX follows:

|WRITE ,<Address to store string at>,<String to be stored/(@)String Variable>

The string must be previously assigned to a variable for the 464, which must then be preceded by @ when used here.

|READ ,<Address to read string from>,<(@)String Variable or (@)Array element for result>[,<Length of string to be read>]

The String Variable must exist before use here; ie it must have been assigned a string with length equal to or greater than the number of characters to be read from memory. In the 464, it also must be preceded with @. It is not possible to use the command form of MID$ as a parameter since this is always interpreted as its function form.

If present,the 'Length of string to be read' parameter will be compared with the length of the result string & the shorter of the two used.If the Length is shorter than the string, only 'Length' number of characters will be read from memory; the rest of the string will remain as before. If the string is shorter or if this optional parameter is omitted, the whole of the string will be read into.

In the final article in the series, we shall be looking at the way that RSXs can be installed in ROMs.

One of the major introductions to the library are the demos. To be blunt, demos are totally useless, but they're wonderful to watch & a delight to listen to. Unfortunately, there are very few (if any) demo writers in the U.K., but in Denmark, the competition between demo writers is strong.

The best of the Danish demos come from New Way Cracking and United Amstrad Crackers. From the demos in my library, N.W.C have the edge. Their demo called FINAL CREATION must be the best demo available for the CPC.

On loading FINAL CREATION, you're presented with the copy-chain.This allows you to copy the demo (it uses a special version of DATA format) but best of all, when you pass it on to your friends, you can leave messages for them which can never be removed!

After the copy-chain, you're presented with a spectacular display of graphical genius. The screen is enlarged to use the whole monitor, with the top half showing part of a face & the bottom half showing a massive multi-colour animated scrolling message. A fantastic tune blasts out from the built-in speaker & this adds atmosphere to the demo.

If I've not yet convinced you to get some of the demos, then remember that it is extremely difficult to describe the quality of graphics & sound in words. They really have to be seen to be believed.

Most people will agree, Stop Press is an excellent D.T.P. package. But, it does have it's limitations. One of these is the size of fonts that can be used. As fonts can only be designed on a 16 by 16 grid, enlarging them to even double size causes a blocky effect.That got me thinking, what about making a font from cut-outs? Sounds sensible, doesn't it? So I sat in front of the computer & designed a very large font to be used for headlines & such.The result was a smooth felt-tip font that was a 100 times better than a normal enlarged one.Of course, this would not fill up a disk, so I also designed 8 new normal fonts (including two small fonts as an alternative to the Amstrad one) and a page of clip art - containing some digitized pictures - to go along with it. The result is PD DISK 79, the ultimate 'add-on' for Stop Press. Disks soon to be released include Art Disk 2 and Hack Attack which contains 50 pokes & a Multiface Poke DataBase complete with details of over 160 multiface pokes. Adventures Five is also due for release soon & so is Applications 3. Also look out for a game produced by Glenco Software using Sprites Alive.This game will be reviewed by Amstrad Computer User and Scull PD Library will be the first to have it. In fact it should be ready by the time you read this.

On the next page is a very concise stock list and details on how to order, as well as a mini-order form (you can photocopy it if you don't want to hack your valuable copy of this mag!). These should be read carefully, but if you are unsure of anything, feel free to give me a call on (03552) 24795, Monday to Friday, 7pm to 11pm only.

PD DISKS LIST - SEPTEMBER 1990 - SCULL PD LIBRARY

01 Serious 1
02 Games 1
03 Games 2
04 Animations 1
05 DW 1
06 DW 2
07 Applications 1
08 Games 3
09 Serious 2
10 Games 4
11 AI/Education/Gfx
12 Games 5
13 Animations 2
14 Art Disk 1
15 Adventures 1
16 Geno Adventure
17 Applications 2
18 Games 6
19 Adventures 2
20 CPM Misc 1
21 Serious 3
22 Quiz 1
23 CPM Applications 1
24 CPM Applications 2
25 CPM Applications 3
26 Games 7
27 Mini Prop (CPM)
28-33 Demos 1-6
34 Mag Indexs 85-87
35 Phone Codes
36 Revision Aid
37 CPM Assemblers
38 CPM Adventure
39 Demos 7
40 Demos 8
41 CPM Games
42 Demon PD Disk 1
43 Adventures 3
44-61 Demos 9-26
62 Over 18's Demos
63 Vidi Digi Pics
64 Adventures 4
65 Serious 4
66 Dazzlestar CPM
67 CPM Languages 1
68 CPM Languages 2
69 CPM Languages 3
70 CPM Applications 4
71 Introduction
72-74 JRT Pascal (CPM)
75 M Basic (CPM)
76 CPM Applications 5
77 Adv Art Studio files
78 Games 8
79 Stop Press Clip Art
80 Yet more demos 28
81 Serious 5

NB: Disks 72-74 must be ordered together. 62 requires proof of age. Disks 28-33 and 44-61 can be ordered separately and are listed together to save space

PD is available on both 3" and 5.25" disks. The latter format is only compatible with drives that use 80 tracks, even though the CPC uses the first 40 tracks only, & either 1 or 2 sides can be used - in normal 178K or 169K formats.You can get any of the above disks by sending £1.00 per disk, the correct amount of disks and an SSAE or you can receive full details by sending a disk and SSAE. All of the above selections require ONE side of a disk only. Also enclose a blank disk for the stocklist disk mag. All orders should be send on the form opposite.

Remember that Rebound and Bandit (2 NON-PD games reviewed in Issue 4) are still available. They cost £5.00 (for both) if I supply the disk, or for a copy-charge of only £1.00 each if you supply the disk. Rebound is a break-out clone with a difference, there is not bat! - "a most professional piece of software" P-O Issue 4. Bandit is a fruit machine game with 5 mini-games! - "If you are the gambling type then this is surely the game for you" P-O Issue 4. Full details can be found on the stock list disk mag.

ORDER FORM

I would like to order the following selections :-

Pd disks ___, ___, ___, ___, ___, ___, ___

Tick here for Rebound and Bandit ___

Name _____

Address _____

_____

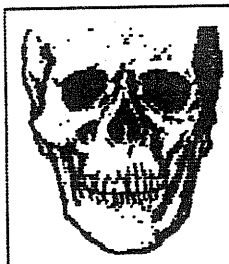
_____

Telephone number _____

Number of disks enclosed _____

Money enclosed _____

Fill this in and send it to the address below



SCULL PD LIBRARY

119 Laurel Drive, East Kilbride, Glasgow G75 9JG. 03552 24795

THE UK'S BIGGEST & BEST!

BASIC M/CODE

POKING AROUND

A Selection of Useful Tips

The two main languages on the CPC, BASIC and Machine Code, both have their advantages and disadvantages. While Machine Code provides many features that are unavailable in BASIC, it's difficult to learn & complicated to program. It would be very useful if we could access these bits of 'Machine Code Magic' from within our own BASIC programs but unfortunately this is almost impossible in nearly all cases.However, in the CPC's memory, there are certain addresses that contain the various bits of information that are necessary for the computer to run correctly. It's a very simple matter to change the data stored in these addresses from within BASIC, via the POKE command. The only drawback is that you're very limited as to what you can achieve and that the results are by no means certain. Therefore, printed below, are various snippets of information (& almost all are in the form of 'pokes') that I have come across over the years. Whilst none of them are particularly exciting, they all do things which would either be much harder or even impossible using BASIC alone.

Un-erasing files

I'm sure that we all know the terrible feeling when we accidentally erase a disc file that we had intended to keep. Well, help is at hand. When you erase a file, it is not immediately removed from the disc but instead is stored in user area &E5. The computer then records that the disc space where this file used to be is now free.

User Area &E5 is illegal and is not normally accessible but it is possible to trick the CPC into letting you use all 256 areas by poking the address &A701 with the desired user number. This allows you much greater flexibility over disc organisation. The side effect is that if you poke &A701 with &E5 (229), you can load any files which have been erased.

Unfortunately, because the space that was filled with the 'erased' file has been recorded as being free, any program which you save may occupy this space on the disc. This will also not retrieve files on a disc that has been reformatted.

Removing Spaces

Here's a quick little poke of varied use. If you're one of those people who like to insert extra spaces in lines (eg. 10 PRINT "Hello") and then wish that you hadn't included them, possibly to produce the right code in Amstrad Action's Typewriter program, then the answer is to poke &AC00 with &FF. This then removes any extra spaces.

Resetting 'TIME'

The CPC includes a BASIC function 'TIME'. It records the time that has gone since you switched the computer on and can be printed using PRINT TIME. However, there is no way of resetting this clock from within BASIC. But if you enter the following lines, then you will find that it is reset to 0.

```
10 REM Reset TIME for the 464
20 POKE &B187,0:POKE &B188,0
30 POKE &B189,0:POKE &B18A,0
```

```
10 REM Reset TIME for the 6128
20 POKE &B8B4,0:POKE &B8B5,0
30 POKE &B8B6,0:POKE &B8B7,0
```

By changing the numbers that are poked in lines 20 and 30, you can set time to whatever value that you want. If you're interested, the total time that can be recorded using the TIME function is about 166 days !! That's enough for me !!

CAPS/SHIFT Lock

If you've ever wanted to be able to turn the CAPS or SHIFT LOCK on/off in a BASIC program (for example whilst getting a name for the hi-score table) all you have to do is poke the addresses below with &FF to switch it on and &00 for off.

464 CAPS LOCK	&B4E8	6128 CAPS LOCK	&B632
464 SHIFT LOCK	&B4E7	6128 SHIFT LOCK	&B631

Disabling ESC

A query that seems to come up time and time again is, how to stop the ESC key from working. The simplest and best method is to poke &BDEE with &C9. This stops all types of break including the CTRL-SHIFT-ESC. If you want to restore it to normal, poke &BDEE with &C3.

Useful Calls

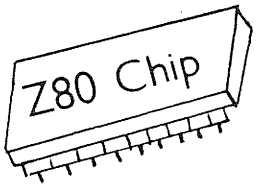
There's one last category of meddling that you can do and that is to access firmware routines direct from BASIC. These are done using the CALL command. The problem is that you are unable to pass parameters to the Machine Code routine & so you are restricted as to what you can achieve. Still, here are some addresses that you may find useful to CALL.

```
CALL &BC02 - resets the colours to those used when first switched on
CALL &0000 - completely resets the computer (equivalent to CTRL-SHIFT-ESC)
CALL &BB03 - clear key board buffer (has some side effects)
CALL &BB06 - waits for a key to be pressed
```

Next issue we'll see if we can drag up some more useful pokes and things to do with your CPC. Remember, if you've any of your own then please send them in.

Machine ....Code....

The STACK



We have now looked at some Machine Code programs which are becoming fairly lengthy and complicated, but there are still one or two common commands that we have omitted. In this issue, therefore, we are going to look at these commands, some of which we have already used but not explained.

In the very first part of this series, we were introduced to the registers (A,B,C,D,E,H,L) and these were likened to BASIC variables except being far more limited. The obvious problem was that they could only store numbers with values from 0 to 255 (pretty small in computing terms). This could be then overcome by forming register pairs (BC,DE,HL) from them - these could then hold values from 0 to 65535 (ie a 16-bit number).

Whilst this is fine in theory, it does have some limitations. Suppose your program needed to store and work on four 16-bit numbers (something which is not unreasonable!), we would find ourselves with a problem - we've got only 4 registers that are capable of holding such a number!

One way around this would be to poke the numbers into a safe place in memory when we weren't using them, and then retrieve them when they were required. This would be a very space consuming exercise, and also very frequent, so we're thoughtfully provided with a couple of commands to do this - PUSH and POP.

PUSH AND POP

PUSH and POP are used to preserve and retrieve register pairs for use later on. The first thing to notice about them is that they can work only on register pairs, and not on individual registers - however, this is not a problem for all we do is PUSH or POP the corresponding register pair instead. For example, if B was to be preserved we would simply use PUSH BC and then to retrieve, it POP BC.

In the previous paragraph, I mentioned a safe place where numbers could be stored. When using PUSH and POP, one area is allocated for this express purpose - the Stack. I have heard the Stack being explained in many ways but one of the best ways of thinking of it is as a pile of boxes. Imagine that each box really represents a memory location, and so has an address. Unfortunately, these boxes have lids on, and so the only one that you can look inside is the very top one. When you give a PUSH command (eg. PUSH HL) the computer takes the number in HL, and puts it in the top box. If you then do another PUSH (eg. PUSH BC), the CPC adds another box on top of the pile and puts BC into it. Now, when you come to POP something off, the only number you can get at is BC.

The Stack is often called a 'last-in, first-out' (LIFO) stack because that is exactly what it is. You can put things on top of it but ONLY at the top, and you can take things ONLY from the top.

To illustrate this, follow the below example through:

```
LD BC,&6789          ; BC = &6789
LD DE,&3453          ; DE = &3453
LD HL,&0012          ; HL = &0012
PUSH BC              ; stack holds &6789
PUSH DE              ; stack holds &3453,&6789
PUSH HL              ; stack holds &0012,&3453,&6789  *
LD HL,0000           ; HL = &0000
LD BC,0000           ; BC = &0000
LD DE,0000           ; DE = &0000
POP HL               ; HL = &0012     stack holds &3453,&6789
POP DE               ; DE = &3454     stack holds &6789
POP BC               ; BC = &6789     stack is empty
```

In this, the left hand number on the stack (*) is the number at the top, & the right most number is the number at the bottom. If you now think you understand the Stack, try and puzzle over this - the Stack is in fact upside-down (ie the bottom of the stack is at a higher memory location than the top!!)
The Stack stretches downwards from &BFFF and could go on for as long as needed (although it would wipe out the jumpblocks and other important bits if it went on for too long).
Fortunately, we don't have to make life complicated for ourselves and, as long as we remember that it is a LIFO stack, we do not need to know which way up it is and where it is located. We can, of course, meddle with the stack by use of the Stack Pointer (SP) - more of that later.

So far, we've just been preserving registers for use at a later date, but we can also use the PUSH and POP commands to do one or two other things. Using a Z80, we cannot do load one register pair with another register pair (such as LD BC,HL), instead we have to go about it in a very long-winded way:

```
LD HL,&7056          ; HL = &7056
LD A,L               ; A = L = &56
LD C,A               ; C = A = &56
LD A,H               ; A = H = &70
LD B,H               ; B = A = &70
                     ; BC = HL = &7056
```

Using PUSH and POP it is much simpler:

```
LD HL,&7056          ; HL = &7056
PUSH HL              ; put the number in HL (&7056) on the top of the stack
POP BC               ; take the number from the top of the stack (&7056) and
                     ; put it into BC, ie BC = HL = &7056
```

There's just one other thing to bear in mind when you are dealing with the Stack and that is that it is not only PUSH and POP which make use of the Stack. The CALL and RET commands also use it extensively. When a subroutine is CALLed, from a Machine Code program, this is what happens: firstly the address that the program should return to, when it has done the subroutine, is put on the Stack. The program then jumps to the subroutine and continues executing the code from this point. When it encounters a RET instruction, the computer takes the number from the top of the Stack (the return address) and goes to that place.

All of this works very well until you start using PUSHes and POPs inside a subroutine. It is then imperative that everything that has been PUSHed onto the Stack is then removed by use of POP. Below is an example of how NOT to do it:

```
        ORG &8000
        CALL hello      ; CALL the subroutine labelled '.hello' The address of
                        ; the LD BC,&C0 instruction is now placed on the Stack
        LD BC,&C0       ; any instruction would have done here
        RET             ; RETurn to BASIC

.hello  PUSH HL         ; put HL on the stack
        PUSH BC         ; put BC on the stack
        LD A,72         ; A = 72 (ASCII for H)
        CALL &BB5A      ; print H
        POP BC          ; take BC off the stack
        LD A,105        ; A  = 105 (ASCII for i)
        CALL &BB5A      ; print i
        RET             ; RETurn from the subroutine
```

Unless you are extremely lucky, this program will cause a crash and you'll have to reset the computer. The reason for this is that, in the subroutine, we put 2 things on the stack and only took 1 off. This left one extra item on the Stack, and the computer took this as the return address, jumped to it, and....Crashed!

There is just one further feature concerning the stack, the Stack Pointer. This keeps track of where the top of the Stack is. Every time we PUSH something onto the Stack, the Stack Pointer is decremented (remember the Stack is upside down) by two (we are PUSHing a two byte register pair) & when something is POPped the Stack Pointer is increased by two. Of course, we can alter the Stack Pointer as if it were a normal register (eg LD SP,&1890 DEC SP etc) and this can produce some rather interesting effects - some of which we will be looking at in a future issue.

As you can see, the Stack, PUSH and POP are very useful items, without which it would be even harder to program in Machine Code. They can also be pretty lethal if not used correctly - so when using the stack, make a note of what you've put on it and then removed. That way you cannot go wrong!

PRINT-OUT'S ASSEMBLER

As promised last issue, we now have our very own assembler which will allow you to enter all of the Machine Code programs that are printed in this magazine, without having to resort to using an expensive alternative.

As we have now got an assembler, we do not intend to print the BASIC Poker numbers in future issues. However, because of the rather extensive documentation which needs to accompany the program and also its length, we have been unable to include it in the magazine (it would have taken up about ten pages). Instead, we have put it, and full instructions for its use, on this issue's program tape and disc.

Our assembler has been designed to be, as near as possible, compatible with Maxam (widely regarded as the standard for assemblers on the CPC) and you should be able to use it with any other magazines' Machine Code programs. The assembler uses a common method of entering assembly language lines - namely to put them in a BASIC program, preceded by a short REM (') and then assembling them through an RSX. This avoids the need to include a memory hungry 'Text Editor' and makes the program both powerful and flexible. This also provides the added benefit that we can supply any Machine Code listings on the program tape/disc as a BASIC program which can then be assembled (thus saving you typing).

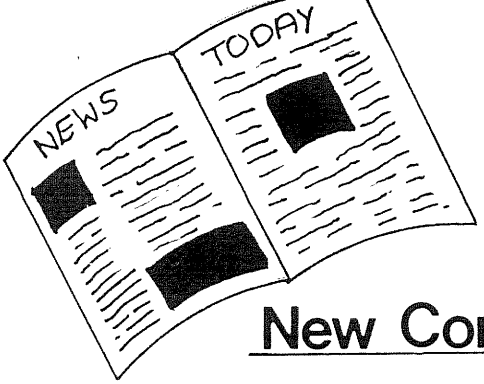
This system is as friendly as is possible for a Machine Code Assembler, and it tells you of any errors in the way that the code has been written. Due to the nature of the system it needs an 'END' directive to be included so it knows when to stop assembling. Full details of this are contained on the tape or disc.

Small Ads

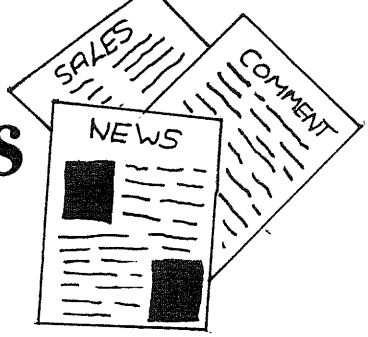
FOR SALE OR SWAP - One SOFT 968 FIRMWARE MANUAL.One SOFT 115 DISC HiSoft Pascal, One DMP 2160 Printer..One 256K Memory Expansion with Software. Ideally I would like to swap for an RS232 Interface and Modem but I'm also willing to sell. Also wanted a good C.A.D. System. Terry Gipps, 501 Long Riding, Basildon, Essex, SS14 1JW

WANTED - Issue One of Amstrad Computer User and issues April 1989 onwards. Will pay good price (especially for Issue One) or will swap for as much PD as deemed reasonable. Alan Scully, 119 Laurel Drive, East Kilbride, Glasgow G75 9JG. Telephone (03552) 24795.

FOR SALE - 2 Homebrew programs, Casino Blackjack (a realistic simulation of the gambling game) & Wordsearch (a utility for solving wordsearch puzzles) As reviewed in Print-Out Issue Six. Both programs cost £4.50 together & this includes the cost of a disc. Contact Barrie Snell, 19 Rochester Rd, Southsea, Portsmouth PO4 9BA.



News and Views



New Console & Pluses

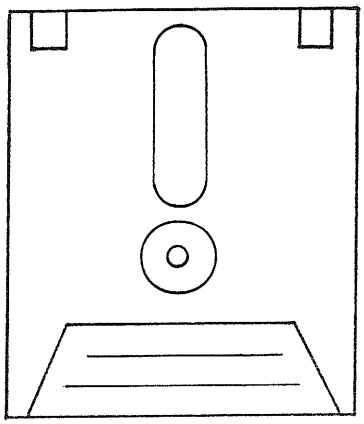
Amstrad has released its new Plus computers and console, thus ending months of speculation by the computer press. Whilst it would appear that a large amount of thought has gone into the design of these new machines, Amstrad seems to have overlooked some rather important factors in their specifications. Firstly, there is no tape interface on the 6128 Plus, thus cutting out almost all of the budget market and removing the option of truly low-cost software. It may be possible to add an external interface at some later date, but it does not seem very sensible for Amstrad not to have done it themselves. Another major problem, the different style of expansion connectors from the old CPC, seems to have been solved by the use of a small plug-in adaptor available from W.A.V.E.

I must say that the custom hardware chip certainly makes the computers look on a par with the Atari ST, although why the sound could not have been more radically improved at the same time, is beyond me. When it comes to existing owners upgrading, I think the most serious problem that Amstrad will face is the worrying lack of information about compatibility between machines. We have been told that 99% of software will work on both the old CPCs and the new Pluses, but what about all of the hardware? I am not prepared to buy a Plus until I know that all my add-ons, ROMS, etc will operate correctly. There's no doubt, in my mind, that Amstrad *could* have a very successful computer in the Pluses, but the question is will they be able to capitalize on it. Only time will tell.

The new console also looks as if it may have a rough time, unless the price of cartridge software drops from the quite staggering £30 that is being asked at present. After all, cheap it may be, but it's up against some fairly stiff competition from the likes of Nintendo.

Spectrum +3 dropped

Also in the last couple of weeks, Amstrad have made what is, in my opinion, a very sensible move - they've dropped one of the Spectrums. The Spectrum Plus 3 is no more, thus making way for the far superior 6128 Plus - its main rival. The news came as a bit of a shock to Spectrum owners but it certainly shows us where Amstrad's priorities lie (at present).However, all of this has been overshadowed by the new 'Generation 3 PCs' which are due for a launch in the near future, and Amstrad are pinning their hopes on the success of these computers to raise their flagging fortunes. After another disastrous year with its audio and video range, Amstrad desperately needs some good news as they have recently announced another downturn in pre-tax profits. Let's hope that Amstrad have finally got everything sorted out !!



DISC NAMER

UTILITY

BY BOB TAYLOR

The purpose of this utility is to provide an identity for each side of a disc whenever it is CATalogued. The idea comes from MS-DOS and DR-DOS (as used on the IBM PC and other computers) where each disc is allowed a 'Volume' name to identify it (both sides of each disc are treated as the one disc).

Of course we have to turn our discs over to access another side, and so it becomes doubly useful to be able to tell which side of which disc we are using. The 'name' written to the disc by this utility is inserted in the disc's Directory area so that it appears on the screen each time we use CAT. In doing so it takes up one of the possible Directory entries leaving 63 available, but never having run out of directory space. I don't see this as a problem for most users.

CHOOSING A NAME

It is possible to have up to ten characters in the name. However, when it is printed by CAT a full stop will appear between the 7th and 8th characters so it is best to arrange your name to fit in with this.

A space is automatically placed before the name when written, with the end result that the title will be the first name printed by the CAT. A further aid to clarity is to use lower case for any letters, to make it obvious that it is not the name of a file.

This utility works only on discs which have been previously formatted with SYSTEM or DATA. The routine will trap such errors as disc missing, no name,name too long, etc. and also allow renaming of sides. First, type in the loader program below (and save it before running it).

When RUN it will prompt you to press 'S' to save the code as 'NAMEDISC.BIN'. The routine is now ready to use (NB &90D bytes of space are needed). For future use just use: `MEMORY &7FFF:LOAD"NAMEDISC.BIN"`

RUNNING THE PROGRAM

The syntax to use for naming a side of a disc is: `CALL &8000,"name"`
However, unfortunate 464 users will have to use : `a$="name":CALL &8000,@a$`
Remember that you will need to name both sides of a disc separately as a CPC treats them as if they were two completely different discs. So simply turn the disc over and then name the other side.

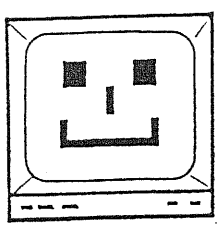
```
[F1] 10 'Disc Namer Loader by R Taylor for PRINT-OUT (Public Domain 1990)
[C1] 20 MEMORY &7FFF:RESTORE 110:PRINT:PRINT"Please wait a few seconds"
[19] 30 FOR lin=0 TO &10D/8-1:total=0:FOR n=0 TO 7:READ a$
[A2] 40 byte=VAL("&"+a$):POKE &8000+lin*8+n,byte
[4B] 50 total=total+byte:NEXT n
[21] 60 READ a$:IF VAL("&"+a$)<>total THEN PRINT:PRINT"Error in line"lin*10+110
        :END
[C4] 70 NEXT lin
[BB] 80 PRINT:PRINT"All M/C loaded":PRINT:PRINT"Press 'S' to save M/C as
        NAMEDISC.BIN":WHILE INKEY$="":WEND:IF INKEY(60)<>-1 THEN SAVE
        "NAMEDISC.BIN",B,&8000,&10D
[88] 90 PRINT:PRINT"To Load and Initialise Disc Namer with a program present
        just Enter:":PRINT"MEMORY &7FFF:LOAD"CHR$(34)"NAMEDISC.BIN"CHR$(34)":
        CALL &8000":PRINT"in Direct Command Mode"
[EA] 100 END
[EA] 110 DATA 3D,C0,21,CA,80,CD,D4,BC,4C5
[AC] 120 DATA D0,DF,C1,80,06,04,3A,51,385
[37] 130 DATA BE,E6,F0,B0,4F,FE,44,16,4EB
[D9] 140 DATA 02,28,02,16,00,1E,00,21,081
[C6] 150 DATA B0,A9,E5,D5,C5,DF,C4,80,5FB
[92] 160 DATA 30,25,06,10,11,EF,01,19,185
[63] 170 DATA 7E,B7,28,1F,FE,E5,28,3B,3C2
[BD] 180 DATA 11,E0,FF,10,F2,C1,D1,E1,565
[8C] 190 DATA 0D,10,DF,21,CB,80,C3,B8,3E3
[C6] 200 DATA 80,21,FC,80,CD,B8,80,F1,513
[3D] 210 DATA F1,F1,C9,E5,21,DC,80,CD,5DA
[D9] 220 DATA B8,80,E1,E5,11,F2,FF,19,519
[66] 230 DATA CD,B8,80,21,F1,80,CD,B8,51C
[65] 240 DATA 80,E1,CD,06,BB,E6,5F,FE,532
[7B] 250 DATA 59,20,DC,11,F1,FF,19,36,3A5
[5D] 260 DATA 00,23,36,20,23,EB,DD,66,2CA
[C8] 270 DATA 01,DD,6E,00,7E,FE,0B,30,303
[36] 280 DATA C0,4F,23,7E,23,66,6F,06,2AE
[1E] 290 DATA 00,ED,B0,EB,7D,E6,1F,D6,4E0
[B6] 300 DATA 1D,30,0A,2F,B7,28,06,47,1B2
[8C] 310 DATA 36,20,23,10,FB,06,14,36,1D4
[EF] 320 DATA 00,23,10,FB,C1,D1,E1,DF,480
[CB] 330 DATA C7,80,11,0C,81,C3,9B,BC,3FF
[8B] 340 DATA 7E,B7,C8,CD,5A,BB,23,18,41A
[D5] 350 DATA F7,6C,C5,07,66,C6,07,4E,3B0
[C9] 360 DATA C6,07,C2,44,69,72,65,63,376
[4E] 370 DATA 74,6F,72,79,20,66,75,6C,335
[95] 380 DATA 6C,0A,0D,00,44,69,73,63,206
[CD] 390 DATA 20,61,6C,72,65,61,64,79,302
[6A] 400 DATA 20,6E,61,6D,65,64,20,22,267
[C2] 410 DATA 00,22,0A,0D,52,65,6E,61,1BF
[7C] 420 DATA 6D,65,3F,00,0A,0D,4E,61,1D7
[9B] 430 DATA 6D,65,20,74,6F,6F,20,6C,2D0
[51] 440 DATA 6F,6E,67,00,00,00,00,00,144
```

THE LISTING.....

Linechecker

A PROGRAM TYPING AID

All programs in Print-Out have Linecheck codes which are enclosed in brackets at the start of a line. Don't enter them in as they're designed to be used with Linechecker to eliminate errors when typing in programs which appear in this magazine. Please note, all programs will run whether Linechecker is being used or not. For information on how to use Linechecker, please see Issue Three.



ADVANCED BASIC ~

BASIC tokens

BY Bob Taylor

The Operating System stores BASIC Commands and Functions in memory (and on Disc) not as a string of letters as we see them when printed on the screen, but by using a system of substitute values, which are called Tokens, & I thought it would be useful to provide you with an extended list of these Tokens and other codes used in the PROGRAM AREA of memory.

I suggest that you confirm for yourselves the information given in this article by using the BASIC program in listing 1.Type in extra lines containing examples of Tokens (using Line Numbers from 10 to 90 only) and then use RUN 1000, RUN or GOTO 1000 as the case warrants. You will notice that the value of every byte is given in Hexadecimal format which gives a clearer display than using decimal.

Listing 1:

```
[44] 1000 line.start=&170
[BB] 1010 line.length=PEEK(line.start)+256*PEEK(line.start +1)
[D6] 1020 line.number=PEEK(line.start+2)+256*PEEK(line.start+3):IF line.number
          >99 GOTO 1070
[CE] 1030 PRINT:PRINT line.number" ("STR$(line.length)")";
[96] 1040 FOR n=line.start TO line.start+line.length-1
[7A] 1050 PRINT" "HEX$(PEEK(n),2);:NEXT
[06] 1060 line.start=line.start+line.length:GOTO 1010
[2A] 1070 PRINT:END
```

When run, the program will print out data for any lines present, which are numbered less than 100. Each line's data will start on a new line and will give the LINE NUMBER in decimal followed by a figure in brackets. This is the LENGTH (also in decimal) of the line. Finally, there will be a sequence of Hexadecimal representations (without the '&' prefix for clarity) of the contents of all the bytes in that line, starting with the length bytes.

The LINE NUMBER is obtained from the third and fourth bytes of the line & the LENGTH from the first and second bytes by using the method explained below for calculating numbers from two byte values. Incidentally, while you can only type in 255 characters for each program line, it is possible to have some line lengths much longer than this (304 bytes is the maximum since this is the size of the LINE INPUT area for tokenised lines) and the reason is given below. The last byte in each program line is the END OF LINE MARKER, byte &00.

Tokens are used to store such things as COMMANDS & FUNCTIONS in a program in a much more compact form; compact because a Token usually only occupies one byte of space compared to 9 characters required to write some COMMANDS. Tokens are also used to indicate the different types of VARIABLES and even to express various ranges of Numbers but often in these cases more space is required than in the original text (this is the reason for the longer line lengths).Here, as in the case of Commands and Functions, the real gain is in the speed of handling of BASIC, when RUN, since the line is in a more 'digestible' form for the Operating System.

I have split the Tokens into two tables. Those in the second list are all Functions and must be preceded by a byte of &FF. The first table is by far the most extensive and contains the Tokens for all the COMMANDS,for VARIABLES, for NUMBERS and for SEPARATORS and DELIMITERS. Also included here are three Functions: 'MID$' and 'FN' which also double as Commands but use an identical Token for both forms; and 'ERL' which is only a Function. None of these three have a preceding &FF byte in their Function usage.

A few Command Tokens are preceded by a byte of &01 and another's followed by &00; these bytes are inserted automatically by the BASIC Editing routine as the line concerned is ENTERed into the program, but their presence will not be seen when the line is LISTed or EDITed.

Note the coincidence in the use of &C9 to represent BASIC's RETURN command as well as being the code for the Assembly Language 'RET' mnemonic.

THE BASIC PARSER

Six Tokens are used to indicate the different types of Variables and these have values of &02,&03,&04,&0B,&0C,&0D to show whether the Variable is Integer, String,Real,DEFINT,DEFSTR or DEFREAL/undefined respectively. There then follow, not tokens,but two distance bytes to facilitate speedy location of the required variable in the 'VARIABLES AREA' in memory. Upon ENTERing a line these are left blank and contain &00 and &00.However, after this section of the program's been RUN, they usually contain the distance from three bytes before the start of the Variables area (which is the END OF LINE MARKER at the end of the last line of the Program) to the first character of the name of the Variable where it's stored in the Variables Area.Having the distance already calculated will save time whenever this part of the program is encountered again. The distance is stored with the low byte first, high byte second in the standard way.

The BASIC Parser (that part of the Operating System which scans along each program line at run time and puts the Instructions there into action) takes any 'distance' value other than &0000 and uses it without checking its validity (do not try poking Variable distance bytes). If &0000 is present, because that part of the program has not yet been used, then the Variables Area is searched from the beginning for the required Variable and when found, the correct distance is stored in these distance bytes. There are some exceptions to this, however; eg. the distance bytes of a non-existent variable cannot be calculated of course so if an Instruction just refers to a previously unused Variable without requiring

it to be established (in the case of PRINT) then the distance bytes will remain empty.

Like the handling of Line Numbers (see later) Variable distances are reset (to the original &00 and &00 in their case) on amending any line of a program. Following the distance bytes come the actual characters of the Variable's name exactly as typed in, lower or upper case, letter or digit, except that the last character of the name has its bit 7 set, ie. it has &80 or 128 decimal added to it. There can be up to 40 characters in a name (letters, digits, "." and "#" although the latter can only be the first or last char and is ignored anyway). All the above also applies to Control Variables (those used with the FOR command) which will appear indistinguishable from normal Variables.
Similarly with the Variables which are used as Function Names with DEF FN's and FN's, although it should be pointed out that in the Variables Area, Function Name Variables exist separately from any normal Variables with the same names.

ARRAYS

At first sight it's easy to mistake a tokenised Array for a Variable; there is a type byte (using the same Tokens as for Variables) followed by two distance bytes, then the Name characters, again as for Variables). Already, however we've missed one difference - the distance bytes now give the distance from five bytes before the Arrays Area (cf. with that for Variables area) to the first character in the name of the required Array where it is stored in the Array area.Quite why the distance reference points chosen for Variables & Arrays should be where they are, is beyond me.The other big difference is that the name of the Array (in the program area) is followed by one or more subscript numbers separated by commas & enclosed by brackets; the forms of these numbers are as given below.

RSXS

The special symbol '|', used for User Commands like RSXs, has a byte of &00 inserted following its ASCII when used in this way, giving the sequence &7C &00. Following this, come the actual characters (converted to upper case) & digits of the RSX name, with the last one having bit 7 set.Any parameters which follow are separated by commas (ASCII &2C), the parameters themselves being Variables (Number or String), Strings (6128 only) and Numbers.

NUMBERS

The method of representing Numbers with tokens varies considerably, as some Numbers are wholly represented by a discrete Token while others have one of several common Tokens followed by the value required in 1,2 or 5 byte form. But one thing to notice about all Number representations,in the Program Area at least,is that they are all positive - if a negative Number is required then a byte of &F5 will precede the Token of the Number concerned which will itself be positive.

1) INTEGERS from 0 to 9: these are wholly represented by Tokens &0E to &17 respectively with no following value bytes; ie to calculate the Number just subtract &0E (14 decimal) from the Token (eg &13 - &0E = 5)

2) INTEGERS from 10 to 255 decimal:these have the common Token &19 followed by a single byte containing the value; eg &19 &0A would be 10 decimal. This is one of the few times where it would be easier to use decimal format for the contents of the value byte to get a clear understanding of the exact correlation between the value of the byte and the number represented.)

3) INTEGERS from 256 to 32767 decimal: the common Token here is &1A followed by two bytes containing the Number; eg 256 decimal would be present as &1A &00 &01, and 257 as &1A &01 &01. When two byte values are used by microprocessors such as the ZILOG Z80A used in the Amstrad the least significant byte (LSB) comes before the most significant byte (MSB). The value is calculated from LSB + 256 x MSB.

4) INTEGERS which are less than -32767, greater than 32767, or any non-integers (ie. those which have anything other than 0 to the right of the decimal point): The common Token is &1F, followed by four bytes which hold a part of the Number called the Mantissa and these bytes are followed by 1 byte holding another part of the Number called the exponent.An explanation of this five byte value format is very involved and really beyond the scope of this article, but I am sure the Editor will oblige with space in a later Issue if some of you would like one.In the meantime here are a few actual examples you could meet:

&1F &00 &00 &00 &00 &7F = 0.25
&1F &00 &00 &00 &00 &80 = 0.5
&1F &00 &00 &FF &7F &90 = 65535
&1F &00 &00 &00 &00 &91 = 65536
&1F &00 &80 &00 &00 &91 = 65537

Negative floating point Numbers are never found in the Program Area but could be present in the Variables Area as a value of a REAL variable. Such REAL variables can only contain Numbers expressed in 5 byte form, so even integers from -32767 to 32767 will also be stored in this way & not in the various formats applicable to the Program Area).

NON DECIMAL NUMBERS

1) BINARY NUMBERS, which we type into our programs preceded by &X have the Token &1B followed by 2 bytes containing the Number (converted to Hexadecimal):
eg &X1100100110001111 gives &1B &8F &C9
2) HEXADECIMAL NUMBERS have the next Token &1C followed by the 2 byte Number;
eg. &FF00 would be &1C &00 &FF.

LINE NUMBERS

By this I mean not those at the beginning of each line but those associated with Commands such as GOTO or GOSUB etc. Line Numbers used in this way have one of two Tokens allocated depending upon whether the program has been run or not.

1) When a Program Line is typed, the Token used is &1E, which is followed by the Line Number itself in two byte form. This is what the BASIC Parser encounters the first time such a Command is met -

2) At this point, the whole program is searched for the required line & on finding it, the address of the byte before this wanted line is stored in place of the Line Number we originally typed in. Also, the Token before these address bytes is changed from &1E to &1D. On encountering this Command a second time, the address is used straight away for the purpose intended without having to search the program all over again, thus making Locomotive BASIC even faster. There is no danger of a false address being left when the program is amended because all the Line Numbers are reinserted in place of addresses before any alterations can be implemented; ie before a line is DELETEd or ENTERed.

However, the BASIC Operating System seems to be somewhat inconsistent in its treatment of all the Instructions which take Line Number parameters. It changes some Line Numbers to addresses but not others.For example, ELSE, GOSUB, RESUME, RUN and THEN are always changed; but AUTO,DELETE,EDIT and LIST aren't.

The rest vary from one another; GOTO is usually changed, but not after an ON ERROR or ON <expression>; RENUM only changes Numbers that occur as actual Lines in the Program before renumbering - even the STEP parameter is treated in this way.

NUMBERS IN DATA STATEMENTS

Parameters after a DATA statement can be strings or numbers. An all digit entry without enclosing quotation marks could be a number or a string, even though it looks like a number. Evidently, it would be awkward for the ENTERing routine to treat all entries that look like numbers as numbers, convert them to one of the forms detailed above and then for the BASIC Parser to find that it was a string after all. So each digit in a DATA parameter is present in its ASCII code form.

MISCELLANEOUS

1) Whenever a program line is ENTERed, a byte of &00 is added to the end of the line we have just typed, before it is inserted into the program; this then acts as an End of Line marker.

2) Between statements in a line we type a colon as a separator. This is altered to a byte of &01, when the line is ENTERed. When LISTing a line, this &01 Token is printed as a ':'.

3) The comma and the semi-colon are both used as separators between PRINT items and in INPUT statements and the comma is also used between parameters with many Instructions; in all such cases these will appear as their ASCII codes (&2C and &3B respectively)

4) The SPACE character (eg after Commands with following numbers) appears as its ASCII (&20), although some spaces are deleted as superfluous on ENTERing a line; eg that after the Line Number at the start of a line. The LISTing routine prints a space after the Line Number automatically. If we type in two spaces after the Line Number then the first one is deleted but the second is still present & will be printed as a second space.

5) Quotation Marks (") are used to delimit Strings & appear as their ASCII (&22).

6) Opening and closing Brackets & the Hash - (,),# - are used in connection with numbers and the brackets also with Functions and these three also appear as their ASCII codes (&28, &29 and &23 respectively).
7) The Amstrad has 256 printable characters and in theory any of them may occur after a 'REM' or ''' (short REM), and any, except for the Quotation Mark itself (which would end a String anyway) may occur inside Quotation Marks, each as its ASCII code. However, the NUL character (ASCII &00), as well as being impossible to copy, would throw a spanner in the works of the LISTing routine if it occurred under these circumstances, and thus resulting in apparently shortened lines and sometimes strange Line Numbers. Any line, containing such a NUL, brought to the screen with EDIT and then re-ENTERed will be permanently shortened. Any line with three consecutive NUL's will cause the Program to END at the NUL's. This is because the Parser takes the first 'NUL' to be the end of a line, and checks the following two bytes to see if they are a valid line length for the next line; a 'length' of &0000 signals the end of the Program (no matter how much longer the Program actually is) and the Parser then returns control to Direct Command Mode with the usual 'Ready' message. However bytes of &00 can occur in a line (eg as a Variable distance or as Number value bytes) without having these effects.

Any formatting characters used with DEC$ or with the USING qualifier for PRINT, will also appear as their normal ASCII's since they too are enclosed by quotes. In a DATA statement, in the case of delimited strings the above applies. However if quotation marks are not used, then, with the exception of the quotes character again, only those characters between [SPACE] and the shaded character with ASCII code &7F or 127 decimal may be used.Any Control Code characters with ASCII's below &20 that are typed in will be replaced with spaces, and any characters with codes above &7F will be deleted.

As any bytes found in a Program line could be a Token or part of a distance, address, value or name they need to be interpreted in the context of the line & not taken at their immediate face value.

I hope that these gleanings will be of help and interest to you and give you a clearer insight into the workings of our excellent computers.I only wish that someone else had published them at the introduction of the CPCs & saved me some of the time I have spent; it's not as though there's something Top Secret about them but the makers of Arnold have played their hands very close to their chests and such information is not easy to come by. On the other hand, I must admit to having enjoyed the effort of finding out some of the foibles and secrets of the Amstrad.In the next issue of PRINT-OUT, I'll be presenting an RSX (based on the information presented here) that can be used to find occurrences of any part of BASIC in a program.

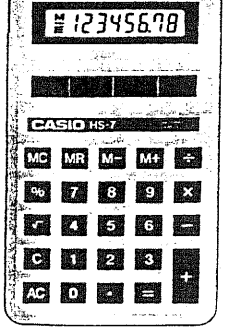
TOKENS TABLES

00 End of Line marker
01 : (Statement separator)
02 % variable (Integer)
03 $ variable (String)
04 ! variable (Real)
05 to 0A NOT USED
0B DEFINT variable
0C DEFSTR variable
0D DEFREAL or undefined variable
0E 0 (number)
0F 1 (integer numbers)
10 2 " "
11 3 " "
12 4 " "
13 5 " "
14 6 " "
15 7 " "
16 8 " "
` 9 " "
18 NOT USED
19 10 to 255 integer numbers (the value contained in the next byte)
1A 256 to 32767 integer numbers (the value contained in the next two bytes)
1B &X (binary numbers; value is held in the next two bytes)
1C & (hexadecimal numbers; value contained in the next two bytes)
1D Program Line Number (converted to the 'address' before the start of the line & contained in the next two bytes;found when this part of the program has already been run)
1E Program Line No. (still as a Line No since this part of the program has not yet been run; value contained in the next two bytes)
1F integers less than -32767 or greater than 32767, and floating point numbers (the value is held in the next 5 bytes)
20 Space - used as separator between parts of a statement
21 NOT USED
22 Quotation mark '"' is used to delimit a string
23 Hash '#' for Windows and Streams
24 to 27 NOT USED
28 opening bracket '('
29 closing bracket ')'
2A to 2B NOT USED
2C comma ',' used as a separator in PRINT items & between parameters
2D Hyphen '-' used with DEFINT etc
2E to 3A NOT USED
3B Semi-colon ';' is used as a separator for Print items etc
3C to 7B NOT USED
7C,00 ¦ (symbol that precedes an RSX command.The &00 byte is inserted when the program is stored & will not appear on listing the program)
7D to 7F NOT USED
80 AFTER
81 AUTO
82 BORDER
83 CALL
84 CAT
85 CHAIN
86 CLEAR
87 CLG
88 CLOSEIN
89 CLOSEOUT
8A CLS
8B CONT
8C DATA
8D DEF
8E DEFINT
8F DEFREAL
90 DEFSTR
91 DEG
92 DELETE
93 DIM
94 DRAW
95 DRAWR
96 EDIT
01,97 ELSE (the &01 byte is inserted in the program when stored & will not appear when listed)
98 END
99 ENT
9A ENV
9B ERASE
9C ERROR
9D EVERY
9E FOR
9F GOSUB or GO SUB
A0 GOTO or GO TO
A1 IF
A2 INK
A3 INPUT
A4 KEY
A5 LET
A6 LINE
A7 LIST
A8 LOAD
A9 LOCATE
AA MEMORY
AB MERGE
AC MID$ (Command, and Function but without preceding &FF byte)
AD MODE
AE MOVE
AF MOVER
B0 NEXT
B1 NEW
B2 ON
B3 ON BREAK
B4 ON ERROR GOTO 0/ON ERROR GO TO 0
B5 ON SQ
B6 OPENIN
B7 OPENOUT
B8 ORIGIN
B9 OUT
BA PAPER
BB PEN
BC PLOT
BD PLOTR
BE POKE
BF PRINT
01,C0 ' (abbreviated version of REM; see note with ELSE Re &01 byte)
C1 RAD
C2 RANDOMIZE
C3 READ
C4 RELEASE
C5 REM (written in full)
C6 RENUM
C7 RESTORE
C8 RESUME
C9 RETURN
CA RUN
CB SAVE
CC SOUND
CD SPEED
CE STOP
CF SYMBOL
D0 TAG
D1 TAGOFF
D2 TROFF
D3 TRON
D4 WAIT
D5 WEND
D6 WHILE
D7 WIDTH
D8 WINDOW
D9 WRITE
DA ZONE
DB DI
DC EI
DD FILL *
DE GRAPHICS *
DF MASK *
E0 FRAME *
E1 CURSOR *
E2 NOT USED
E3 ERL (Function only; no preceding &FF byte)
E4 FN (Command when used with DEF Token and SPACE,and Function but without preceding &FF byte)
E5 SPC
E6 STEP
E7 SWAP
E8 to E9 NOT USED
EA TAB
EB THEN
EC TO
ED USING
EE >
EF =
F0 >=
F1 <
F2 <>
F3 <=
F4 +
F5 -
F6 *
F7 /
F8 ^
F9 \
FA AND
FB MOD
FC OR
FD XOR
FE NOT
FF Function prefix (see Table 2)

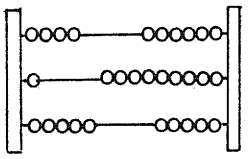
The CPC 464 does not have certain of the above Commands.They are indicated by a '*',nor does it have certain combined commands, eg:- CLEAR INPUT, GRAPHICS PEN, GRAPHICS PAPER.It does however have MID$ as an undocumented command. The 464 also does not have the two functions DERR and COPYCHR$. It also does not perform DEC$, even though it recognises the function.

Functions (preceded by &FF)

00 ABS
01 ACS
02 ATN
03 CHR$
04 CINT
05 COS
06 CREAL
07 EXP
08 FIX
09 FRE
0A INKEY
0B INP
0C INT
0D JOY
0E LEN
0F LOG
10 LOG10
11 LOWER$
12 PEEK
13 REMAIN
14 SGN
15 SIN
16 SPACE$
17 SQ
18 SQR
19 STR$
1A TAN
1B UNT
1C UPPER$
1D VAL
1E to 3F NOT USED
40 EOF
41 ERR
42 HIMEM
43 INKEY$
44 PI
45 RND
46 TIME
47 XPOS
48 YPOS
49 DERR *
4A to 70 NOT USED
71 BIN$
72 DEC$ *
73 HEX$
74 INSTR
75 LEFT$
76 MAX
77 MIN
78 POS
79 RIGHT$
7A ROUND
7B STRING$
7C TEST
7D TESTR
7E COPYCHR$ *
7F VPOS
80 to FF NOT USED



NUMBER SYSTEMS



~ 2'S COMPLEMENT

The only number system that the CPC can understand is binary and the reason for this is that binary involves only ones and zeros which in turn are recorded by the computer as being either 'on' or 'off'. Each digit in binary is called a BIT and eight digits form a BYTE. Thus an 8-bit number has eight digits (each of which can be either 1 or 0) and a 16-bit number has 16 digits.

The lowest value that a standard 8-bit number can hold is zero (when all of the bits are set to zero - ie 00000000) and the highest value is 255 (when they are all set to one - ie 11111111). Likewise a 16-bit number can have a value of between 0 and 65535. The reasons for this were explained in the very first issue of Print-Out.

However there's one very serious problem with this system; how can the CPC store negative numbers? A lot of the time this problem can be avoided but there are certain occasions when it is essential to to represent negative numbers and so 'Two's Complement' or the 'signed number system' was devised.

Two's Complement still uses the binary counting system in exactly the same way as before - it is just the way in which the digits are interpreted that has changed. When using 2's Complement it's only possible to represent numbers from -128 to 127 in an 8-bit binary number. To work out what the negative number is, the following rules must be followed:

First, write out the positive number in binary form:-

eg. 105 = 01101001

Then change all of the zeros to ones and all the ones to zeros:-

eg. 10010110

And finally, add 1 to this. This gives the negative form:-

eg. -105 = 10010111

An important point to remember, when using this method, is that the binary number must always contain eight digits. As proof that the system actually does work, we will add -105 to +105 and see if the answer is correct (ie. zero)

```
        -105    =    10010111
PLUS     105    =    01101001

EQUALS     0    =   100000000
```

There's something not quite right about this answer until you remember that an 8-bit, signed number must contain only eight digits. Therefore, the left-most digit can be discarded thus giving 00000000 which is the correct result. Numbers which have been produced using 2's complement are known as 'signed' numbers, and those that have not are called 'unsigned' numbers.

When dealing with signed numbers, a quick way of telling if it is positive or negative is to look at the left-most digit. If it is a 1, the number is negative, otherwise it is positive.

Here is a brief summary of the differences between signed and unsigned numbers:

8-BIT SIGNED NUMBERS can represent both positive and negative numbers and range from -128 (equals 10000000 in binary) to 127 (equals 01111111).

8-BIT UNSIGNED NUMBERS can represent only positive numbers and range from 0 (equals 00000000 in binary) to 255 (equals 11111111).

The same rules, methods of conversion and differences apply to 16-bit numbers; the only change is that there must be 16 digits in the binary number and that numbers from between -32769 and 32767 can be represented when it is in signed form, and between 0 and 65535 in its unsigned form.

There's only one problem remaining; how does the computer know when a number is signed or unsigned. The answer is that it depends on what type of number it is expecting.For example, in last issue's Machine Code, there was a program to draw lines using relative coordinates. Negative numbers had to be given to the CPC and these were represented as 16-bit signed numbers. As the screen is only 640 pixels wide by 400 pixels high, the computer knows that we are unlikely to want to draw to a place about 65000 pixels along, and so treats it as a signed number, thus meaning about -400 pixels - far more sensible when using relative coordinates. This might not seem much help, but do not worry - the CPC is very good at getting it right !!!

As a conclusion to this article, here's a short program which asks for an 8-bit number (in either decimal,binary or hexadecimal form) and then works out what the negative version of this number is. If you enter numbers greater than 255, it's likely to give results which you may not expect. Binary numbers must be preceded by &X and hexadecimal numbers by the '&' sign. The program expects you to enter a positive number & will then give you the negative equivalent. However, rather interesting and unusual results can be obtained by entering a negative number to begin with.

```
10 REM Two's Complement Converter
20 MODE 2:INPUT "Enter the number: ",num$
40 a$=BIN$(VAL(num$),8)
50 PRINT "The binary number is ";a$
60 FOR i=1 TO 8
70 t$=MID$(a$,i,1)
80 IF t$="1" THEN dig$="0" ELSE dig$="1"
90 r$=r$+dig$
100 NEXT i
110 r$="&X"+r$
120 c=(VAL(r$)+1)
130 com$=BIN$(c,8)
140 PRINT "The complement is: ";com$
```

Whilst signed numbers & 2's complement may seem complicated, all of the difficulties can easily be overcome by use of a good assembler, such as the one on this issue's program tape or disc!!!

Prizes to be Won !!

As this is our birthday issue, we have decided to run a competition to celebrate and have managed to get hold of some wonderful prizes. You may remember that in our last issue we reviewed a copy of Tearaway from CPC Network and it was highly recommended. Tearaway is the ultimate hacking program - it installs itself in a Multiface Two and at a touch of the red button you have a fully fledged monitor, disassembler and a host of other features all at your fingertips. Don't worry if you have not got the necessary set-up to use Tearaway, we have got a set of dust covers (please tell us which computer you have) and a copy of 'Rick Dangerous 2' to give away. In order to have more chances to win, you choose to go for as many of these prizes as you like. First, you have to decide on the correct answer for each of these questions, then just complete the box at the bottom of the page:

1. Amstrad's newly released console is called the....
   a) GX400 b) GX4000 c) XG4000

2. The new Pluses and Console both use the same central processor. Is it....
   a) 6502 b) 68000 c) Z80

3. The piece of cartridge software supplied with the new machines is....
   a) Fire and Forget II b) Batman - The Movie c) Burnin' Rubber

4. In which major European city were Amstrad's three new computers launched....
   a) Paris b) London c) Berlin

5. Who is Amstrad's Technical Manager - he also helped design the original CPC...
   a) Alan Sugar b) Keith Patterson c) Roland Perry

Just ring the correct answers, and then tick which prizes you could receive (the more you tick the more your chances of winning!). Finally, fill in your name and address. Then send your competition entry to us at, Print-Out, 8 Maze Green Road, Bishop's Stortford, Hertfordshire CM23 2PJ.

Name : .....
Address : .....,
.....
.....
Postcode : .....

I'd like to win (tick as many as you want):

TEARAWAY (6128 and Multiface only)
DUST COVERS for the 464/6128/Colour/Mono
RICK DANGEROUS II (on tape)

TO INCREASE YOUR CHANCES OF YOU WINNING A PRIZE WHY DON'T YOU SEND US THE COMPLETED QUESTIONNAIRE OVERLEAF AT THE SAME TIME ?

PRIZE QUESTIONNAIRE

In Issue One, we included a prize questionnaire so that we could find out what sort of things people wanted from the magazine. Times change and a year in computing is a long time, so we thought that it would be an idea to run another questionnaire to discover what our readers want in the next year. To provide an incentive for sending in your questionnaire, we're going to give away a copy of

PROTEXT (tape)

to the first person whose questionnaire is drawn from the hat. So, remember to write your own name and address clearly on the back of this sheet. The address for all questionnaires is PRINT-OUT,8 Maze Green Road,Bishop's Stortford,Herts.

1. Which pieces of hardware (including computer) do you own ?

2. What program/game do you use the most frequently ?

3. Which area(s) of Print-Out interest you most ?

4. Which area(s) of Print-Out interest you least ?

5. Do you want more, less or the same of the following articles ?

BASIC Programming	MORE	SAME	LESS
M/Code Programming	MORE	SAME	LESS
Homebrew Reviews	MORE	SAME	LESS
Public Domain	MORE	SAME	LESS
Using CPM	MORE	SAME	LESS
Solutions to problems	MORE	SAME	LESS
News and General	MORE	SAME	LESS
Type in programs	MORE	SAME	LESS

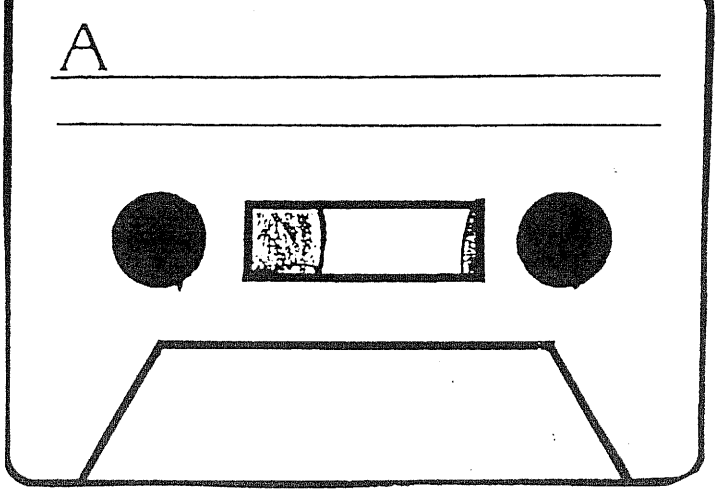
6. Do you find the articles too technical/too simple/about right ?

7. What article(s) would you like to see in future issues of Print-Out ?

8. What do you feel about Amstrad's new computers and why ?

If you have any further comments please feel free to write them on another piece of paper. Remember, what you think and want is very important.

Offers



Please make all cheques payable to Print-Out but any postal orders should be made out to T J Defoe as this saves the Post Office a great deal of time and effort. Unless it cannot be avoided, it is advisable not to send cash through the post. All orders should be sent to :- PRINT-OUT, Special Offers, 8 Maze Green Road, Bishop's Stortford, Hertfordshire CM23 2PJ.

ISSUE EIGHT

If you wish to order a copy of Issue Eight in advance you may do so by sending a cheque / postal order for £1.10 (or 70p + an A4 SAE with a 28p stamp) to the usual address. We hope to have it published by about the 30th November, and it will be forwarded to you as soon as it is available. You may also order a copy of the program cassette or disc in advance by sending the correct amount.

PROGRAM TAPES AND DISCS

We supply both program tapes and discs for all the issues and the prices given below also include a booklet to explain how the programs work plus postage and packing. Tapes and discs are available for Issues One, Two, Three, Four, Five, Six and Seven. The cost for the program tapes are:-

a) A blank tape (at least 15 minutes) and 50p (p+p)
or b) £1.00 (which also includes the price of a tape)

And the cost for a program disc is :-

a) A blank formatted disc and 50p (p+p)
or b) £3.00 (which also includes the cost of a MAXELL/AMSOFT disc) *

BACK COPIES

We still have a supply of Issues One, Two, Three, Four, Five and Six available and the price is £1.10 which includes postage and packing. Alternatively, you can order both a back issue and its corresponding tape or disc by sending:-

a) £1.75 - includes the tape, the required issue and postage and packing
b) £3.75 - includes the disc (genuine MAXELL/AMSOFT disc) * and also the required issue and postage

* When ordering using this particular method please allow about 21 days for delivery as we must rely on outside suppliers for the discs.
* Please also note that one side of one CF-2 disc will hold all the programs from upto six issues. Therefore, the cost is £3.00 for a disc plus one set of programs and then 50p for each additional issue thereafter.